Innovations in Systems and Software Engineering manuscript No.
(will be inserted by the editor)

Maintaining the Health of Software Monitors

Suzette Person · Neha Rungta

Received: date / Accepted: date

Abstract Software health management (SWHM) techniques complement the rigorous verification and validation processes that are applied to safety-critical systems prior to their deployment. These techniques are used to monitor deployed software in its execution environment, serving as the last line of defense against the effects of a critical fault. SWHM monitors use information from the specification and implementation of the monitored software to detect violations, predict possible failures, and help the system recover from faults. Changes to the monitored software, such as adding new functionality or fixing defects, therefore, have the potential to impact the correctness of both the monitored software and the SWHM monitor. In this work, we describe how the results of a software change impact analysis technique, *Directed Incremental Symbolic Execution* (DiSE), can be applied to monitored software to identify the potential impact of the changes on the SWHM monitor software. The results of DiSE can then be used by other analysis techniques, e.g., testing, debugging, to help preserve and improve the integrity of the SWHM monitor as the monitored software evolves.

Keywords Software Evolution · Program Analysis · Software Health Management

Suzette Person
NASA Langley Research Center
Tel.: +757-864-6408
Fax: +757-864-4234
E-mail: suzette.person@nasa.gov

Neha Rungta
NASA Ames Research Center
Tel.: +650-604-4628
Fax: +650-604-3594
E-mail: neha.s.rungta@nasa.gov

1 Introduction

The size and complexity of software in safety-critical systems have increased considerably over time, due in part to the addition of richer feature sets, more automation, and continued efforts to improve the safety and reliability of these systems. As a result, the task of verifying and validating these larger and more complex software systems has become much more challenging and time consuming, requiring new techniques to help ensure the reliability[1] and correctness of safety-critical systems.

Software health management (SWHM) techniques have recently been developed to complement the various verification and validation processes applied to safety-critical systems prior to their deployment [12, 29, 37–39]. *Monitors* are at the core of these techniques. SWHM monitors observe and analyze the system in its execution environment during runtime to detect and respond to violations, and to predict possible failures in the near future. These monitors are often implemented as software components, and as a result also require some level of analysis to ensure their correctness and reliability. The analysis of monitors is important because monitors often serve as a last line of defense against the potentially catastrophic effects of faults in safety-critical systems.

Software changes are inevitable in most deployed systems - successful software systems evolve as requirements change and defects are fixed. Even in safety-critical systems, software is rarely exempt from change after deployment. For example, the discovery of a critical defect (bug) in the system may require an update to the operational software to avoid a system failure.

[1] We use the term *reliability* to mean ‘continuity of correct service’ as specified in [4].

Systems also undergo change when new functionality is added. Changing operational software, however, is known to be risky. Even small changes to the code can have a major impact on how the software executes. Moreover, bug fixes may not always fix the defect and can potentially introduce new defects. For example, a recent study showed that 14.8%–24.4% of the sampled fixes for post-release bugs in several large, mature operating systems were incorrect and had a negative impact on the end users [48].

SWHM monitors use information from the specification and implementation of the monitored software to determine which values to analyze and to determine tolerance ranges for those values. This tight coupling of monitor and monitored software means that the impact of a change to the monitored software has the potential to also impact the SWHM monitor and its correctness in the context of the change. Various change impact analysis techniques have been developed to identify the differences between two program versions in order to guide testing and verification efforts on the changed software [1,6,14,17,20,28,36,45]. The objective of these techniques is to reduce the time and cost of testing and verification of the changed system by guiding the analysis towards the parts of the system impacted by the changes. However, to the best of our knowledge, change impact analysis techniques have not been explored in the context of how information about changes to monitored software can be used to help identify the impact of the changes on the SWHM monitor.

In previous work [28,34], we present Directed Incremental Symbolic Execution (DiSE), a change impact analysis technique for computing the effects of program changes in terms of program execution behaviors. The DiSE technique can be used throughout the software development lifecycle, to help guide software engineering tasks such as testing, debugging, and regression verification tasks whenever software changes are necessary. In this work, we explore how the results of DiSE, applied to the monitored software, can be further leveraged in order to maintain and improve the integrity of SWHM monitor software. The main contributions of our work include:

- We describe how the results of the DiSE change impact analysis on monitored software can be used to identify the impact of the changes on the SWHM monitor software.
- We apply our technique to a system with a SWHM monitor modeled as a Bayesian network and evaluate its cost and effectiveness with the following two research questions:
  RQ1: How does the cost of applying DiSE to the monitored software compare with using traditional symbolic execution to compute the impact of the changes?
  RQ2: How does the number of impacted path conditions generated by DiSE compare with the number path conditions generated by traditional symbolic execution?
- We describe how the results of DiSE on the monitored software can be used to help validate and update the SWHM monitor software to preserve and improve its integrity as the monitored software evolves.

2 Software Health Management

The size and complexity of software in safety-critical systems is increasing rapidly as more components are added to facilitate automation. The number of sensors and actuators on aircraft has steadily increased over time, as has the software to control and monitor these devices. More sophisticated algorithms for the autopilot, navigation, collision detection and avoidance, and other on-board systems, have also contributed to the increase in software. In recent times, we have also seen a shift of responsibilities from pilots to automated systems for a large number of tasks. In the next generation of aircraft, we expect to see continued growth in the size and complexity of the software to enable even more automation in these systems.

Rigorous design, verification, and certification processes have been established to check the correctness of safety-critical software before it is deployed. However, the size and complexity of the systems prohibit exhaustive testing and verification. Moreover, it may not be possible to anticipate or re-create particular environmental conditions for verification purposes, and therefore parts of the system may not be tested prior to deployment. In order to address these limitations, SWHM techniques have been proposed to monitor the software after it is deployed.

Building on decades of research in systems and vehicle management, together with research in software runtime verification, SWHM techniques [12,29,38,39] have been developed to support monitoring of software as it executes and interacts with the hardware (sensors and actuators) after deployment. SWHM monitors perform fault detection, isolation and recovery. They can also monitor for assumption violations and other conditions that are useful in post-flight analysis. Software health management software often serves as a guardian to the system during its operational phase, ensuring its correct and safe operation.

Software health managers have been developed to monitor the values of sensors and variables in software, as well as updates to these variables, the health of the sensors and variables, and also to compute the likelihood of failure using Bayesian networks [38, 39]. Software health managers have also been developed to monitor the health of components in the system; detecting anomalies, identifying and isolating the fault causes of the anomalies (when feasible), prognosticating future faults, and when possible, mitigating the effects of faults [12]. Software monitors have also been generated and used in embedded systems with hard real-time constraints, to sample variables in the monitored software and implement fault tolerant algorithms to determine the health of the monitored software [29].

SWHM systems have been implemented at both the model-level [38, 39] and at the code-level [12, 29]. Model-level systems are tested using model simulations on a wide range of sensor inputs and various values for internal parameters. The analysis of the model is intended to provide confidence in the correctness of the expected behavior of the model. Code-level SWHM systems often have the same V&V requirements as the monitored software, e.g., to achieve a particular level of code coverage during testing, and they are expected to satisfy similar verification conditions as the monitored software.

In addition to functional properties, SWHM monitor software is typically expected to preserve certain non-functional properties, such as non-interference and timing properties. For example, SWHM monitors must not interfere with the monitored software, e.g., change the behavior of the monitored software (unless the monitored property has violated a contract). They must also avoid corrupting any data or causing any crashes in the system. SWHM monitors also must not miss violations or alarms, and should minimize the number of false alarms.

The verification and validation of SWHM monitor software, similar to any other software, is an ongoing process that is necessary throughout the development lifecycle to ensure changes to the system have their intended effects and that no unintended behaviors were introduced by the changes. In the case of SWHM monitors, verification and validation of the monitor may also be required when the *monitored* software is changed, due to the tight coupling between the monitor and the monitored software, e.g., through the values that are monitored. Techniques that identify what is changed and the impact of the changes on the SWHM monitor play an important role in maintaining the health (correctness) of SWHM monitor software. Before presenting our technique for maintaining the health of software monitors, we first provide background on software change impact analysis techniques and discuss some of the challenges associated with computing precise software change impact information.

3 Change Impact Analysis

Software change impact analysis techniques [3] are used to detect the parts of a program affected by the changes made to the code. Given the evolutionary nature of software development, these techniques play a critical role in software development and maintenance, where even a one line fix can potentially have unintended and even disastrous consequences. The results computed by change impact analysis techniques have been widely used to support software maintenance tasks, such as regression testing [21, 23, 33, 41], regression verification [5, 40], studying changes in large code bases [32], and for automated generation of program documentation [7].

Given two closely related program versions, change impact analysis techniques are performed in two steps: 1) compute the differences between program versions, i.e., the *change set*, and 2) using the change set as input, compute the impact of the the differences, i.e., the *impact set*. The change set can be computed based on a variety of program representations. Computing changes based on source code is commonly used in practise because it is efficient and automated. The differencing techniques based on textual differences, however, are often sensitive to formatting and syntactic changes that *may not* affect the way the program executes.

Differences computed based on some graphical representation of the code, e.g., Abstract Syntax Tree (AST), Program Dependence Graph (PDG), Control-Flow Graph (CFG), are in general, more precise than differences computed on the source code as is. The graphical representations of code encode additional information about the program, e.g., control and data dependences, that is useful for computing more precisely the impact of source code changes. Any technique, however, that computes change impact based strictly on differences in the source code structure will have limited capabilities to reason about the impact of the changes on the execution of the code. This is especially true when dynamically allocated data and complex control structures such as loops and recursion are present in the program.

Once the change set is computed, change impact analysis techniques compute which parts of the program may be impacted by the changes. The impact of changes can be computed using information from a static representation of the program or using dynamic information obtained through program execution. Techniques such as [2] analyze a static representation of the program to compute the impact set in terms of program statements that may be directly or indirectly impacted by the changes to the source code. Godefroid et al. statically check whether previously-computed symbolic test summaries are still valid, i.e., not impacted by code changes, to support compositional dynamic test generation [13].

Techniques which use dynamic information [20, 33] have the potential to compute more precise impact sets because they are based on actual program execution paths. Dynamic analysis is typically driven using a set of test cases, so the impact sets will be computed with respect to the specific execution paths explored, which may be a small subset of all feasible execution paths. Other recent work has explored the use of symbolic execution results to compute precise change impact characterizations by systematically exploring the program execution space [15, 31, 27, 42, 45]. The results of these change impact analysis techniques have been used to support a range of software evolution tasks, including test case selection and test suite augmentation; however, scalability is an issue for these techniques.

The change impact analysis used in this work, Directed Incremental Symbolic Execution (DiSE) [28, 34], combines the efficiency of static program analysis techniques with the precision of dynamic analysis techniques to compute the impact of software changes in terms of program execution behaviors. This approach results in a more precise impact set than using static analysis alone, and also addresses the scalability issues associated with symbolic execution by using the results of the static analysis to direct symbolic execution towards the parts of the program impacted by the changes. This effectively 'prunes' the program behaviors that are not impacted by the changes to the code. Because the results of DiSE are computed in terms of program execution paths, they can be used to support a range of software maintenance tasks, including regression testing, debugging and regression verification.

In the following sections we describe the DiSE algorithm and illustrate how the impact set for a small working example is computed. We then describe a novel application of DiSE results computed on the monitored software to help maintain and improve the integrity of the SWHM monitor software.

4 Directed Incremental Symbolic Execution

Directed Incremental Symbolic Execution (DiSE) [28, 34] is a program analysis technique for computing the impact of changes to software. The output of DiSE is a characterization of the effects of code changes on program execution behaviors. The effects are characterized in terms of the inputs to the program and the effects of execution on variables in the program. In previous work [5, 28, 34] we describe how DiSE is a general change impact analysis and how the change impact results computed by DiSE can be used for various software maintenance and evolution tasks, including test case selection and prioritization for regression testing, debugging, and regression verification. In this work we describe a novel application of DiSE results to software health management and discuss how the results computed by DiSE can be used to help ensure the correct operation of a SWHM monitor when the monitored software is changed.

The novelty of the DiSE change impact analysis is to leverage the efficiencies of static analysis techniques for computing the impact of program changes to guide a more precise analysis technique, symbolic execution, to explore and characterize program execution paths that may be impacted by the changes. Our work was inspired by Regression Model Checking (RMC), a technique which uses the results of a static analysis to explore the 'dangerous' elements in the state space whose behavior may be impacted by the changes to the code [46]. DiSE differs from RMC in that DiSE is based on incremental symbolic execution, rather than model checking, and DiSE does not require analysis results to be carried forward as the software evolves - only the source code for two related program versions is required.

An overview of the DiSE analysis is shown in Fig. 1. The inputs to DiSE are two related program versions: the original source (S) and the modified source (S'), and a source-level syntactic *diff* between S and S'. The source-level diff provides information about the source code lines that are *changed*, *added*, and *removed* between S and S'. A control flow graph (CFG) for S', shown in Fig. 1, is constructed from the source of S' and used to guide symbolic execution towards impacted program behaviors.

There are two phases of analysis in DiSE. Phase I estimates the impact of the differences on the source code of S'. Phase II uses the information generated in phase I to compute, with better precision, the impact of the changes on the program execution behaviors. The two phases are shown in Fig. 1. The output of DiSE is the set of program behaviors in S' impacted by the differences between S and S'. The set of *Impacted Program Behaviors* can then be used to identify the impact of the changes on the software health manager (*SWHM Monitor*) responsible for monitoring the software, as demonstrated in Fig. 1.

In the remainder of this section we describe each phase of DiSE. In Section 5 we describe how the results

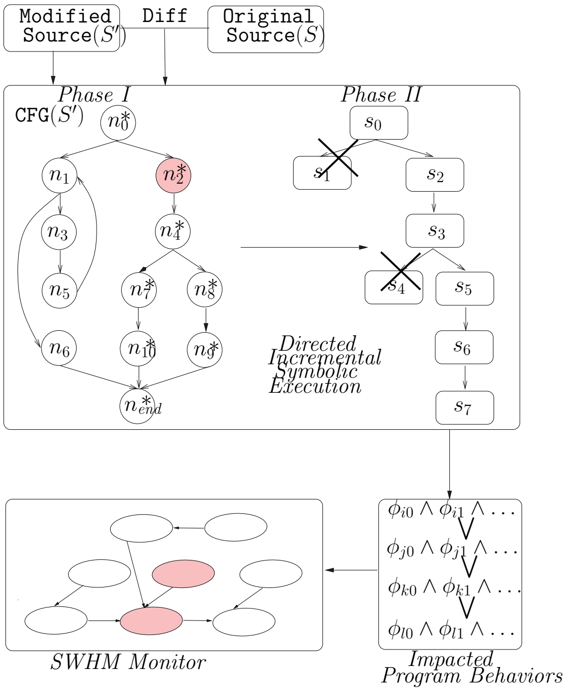
Modified Source(S')
Diff
Original Source(S)
Phase I
Phase II
CFG(S')
n_0^*
n_1
n_2^*
n_3
n_4^*
n_5
n_7^*
n_8^*
n_6
n_{10}^*
n_9^*
n_{end}^*
s_0
s_1
s_2
s_3
s_4
s_5
s_6
s_7
Directed Incremental Symbolic Execution
$\phi_{i0} \wedge \phi_{i1} \wedge \ldots$
$\bigvee$
$\phi_{j0} \wedge \phi_{j1} \wedge \ldots$
$\bigvee$
$\phi_{k0} \wedge \phi_{k1} \wedge \ldots$
$\bigvee$
$\phi_{l0} \wedge \phi_{l1} \wedge \ldots$
SWHM Monitor
Impacted Program Behaviors

Fig. 1 Overview of DiSE. The inputs to DiSE are two syntactically similar program versions, S and S', and a source-level *diff* between S and S'. The output of DiSE is a set of impacted program behaviors that can be used to manage the health of a SWHM monitor.

of DiSE can be used to help identify the impact of the changes on the software health manager.

4.1 Source Impact Analysis

In the first phase of DiSE, information about the added, removed and changed lines of code is used by data- and control-flow analyses to mark additional lines of code that may be impacted by the differences between S and S'. The static analysis computes the impact of the changes by analyzing a control-flow graph representation of the source code. A data-flow analysis is used to identify where variables are defined and used, but the concrete (actual) values of variables in the program that are possible within its environment are not computed. This under-approximation of the execution environment makes the analysis efficient and scalable to larger programs. Furthermore, the analyses in phase I are conservative, i.e., every source line of code that *may* be impacted by the change, will be marked as impacted. This ensures that the source impact analysis does not miss marking any instructions that are impacted by the changes, although it may also mark code as impacted, when in fact, it is not.

Consider the annotated control flow graph (CFG) for the modified source, S' in Fig. 1. Each node in the CFG ($n_0 \ldots n_9$ and n_{end}) represents a single line of source code. Nodes n_0 and n_{end} are the respective entry and exit nodes to the program. All nodes in the CFG are reachable from n_0, and n_{end} is reachable from all nodes in the CFG. The edges between the nodes represent the flow of control between the different program statements during execution. The shaded node, n_2, represents the changed source line of code based on the results of the source-level *diff* comparing S and S'.

During the source impact analysis phase of DiSE, impacted nodes are computed by starting with the set of changed nodes and then using data- and control-flow information to compute the impacted nodes (program statements). In Fig. 1, the nodes annotated with '$*$' (n_0, n_2, n_4, n_7, n_8, n_9, n_{10}, and n_{end}) represent the source lines impacted by the change. The output of this analysis is the set of source lines of code in S' impacted by the differences between S and S'. This information is used to direct the more precise analysis, symbolic execution, in phase II of DiSE. In the remainder of this section we provide a high-level description of the DiSE algorithm. The reader is referred to [28,34] for a detailed description of the analyses implemented in DiSE.

4.1.1 Estimating Impact Based on Control Flow

Conditional branch statements, e.g., `if` and `while`, compare the values of the specified variables and constants and then follow the appropriate branch based on the results of the comparison. Explicit changes to a conditional statement, i.e., changes to the comparison operator or the operands, may impact which code block is executed as a result of the change. For example, consider the following code fragment:

```
   int condTest(int x){
1:  if (x > 0)
2:    return x + 1;
3:  else
4:    return x - 1;
5:  }
```

This code returns $x + 1$ when the input value of x is greater than 0, and when the input value of x is not greater than 0, the code returns $x-1$. The execution behavior of this code can be summarized in various ways. An example summarization based on the program inputs and outputs is: "for any input value of x, the program never returns 0 or 1." Suppose the comparison operator at line 1 is changed to '>='. As a result of the change to the code, the execution behavior of the program is impacted and can now be summarized as—"for any input value of x, the program never returns 0 or -1.” This example demonstrates how a change to the comparison operation of a conditional branch statement impacts the behavior of the program.

Recall that the source impact analysis does not consider the execution environment of the program, i.e., the possible values of x during runtime. Instead, the analysis will *conservatively estimate* that the change in the conditional branch statement will impact *all* statements whose execution are dependent on the result of the comparison operation. For the `condTest` example shown above, when the comparison operator is changed at line 1, the analysis estimates both line 2 and line 4 to be impacted. The analysis considers all changes to conditional branch statements, including the addition and deletion of conditional branch statements, when estimating the impact of the changes on the control flow of the program.

4.1.2 Estimating Impact Based on Data Flow

Changes to an assignment statement in a program may impact the value of program variables. And, as a result, other assignment statements, return statements, and comparison operations that execute after the changed assignment statement and use (read) the changed value may also be impacted by the change. Consider the following code fragment:

```
   int dataTest(int x){
1:  x = x + 1;
2:  tmp = 0;
3:  if (x > 0)
4:    tmp = x + 1;
5:  else
6:    tmp = x - 1;
7:  return tmp;
8:  }
```

In this example, suppose the assignment to x at line 1 is changed to $x = x - 1$. Using a data-flow analysis (which also takes into account the control flow of the program), the analysis would then identify the impacted statements as: (a) the assignment statements at lines 4 and 6 where the value of x is *used* to compute the value assigned to the *tmp* variable, (b) the conditional branch statement at line 3 where the value of x is *read* and compared with 0, and (c) the `return` statement that *reads* the value of *tmp*. Note that the `return` statement is marked as impacted because a transitive closure is computed for the data-flow analysis.

The source impact analysis performed in phase I of DiSE is guaranteed to terminate. In the worst case, all of the source lines in the program are marked as impacted. Such a case would generally be observed for a program that has a very high coupling between its components and variables. Another case is when the change is made to a part of the program that interacts with all of the other parts of the program. In general, we do not expect the small, incremental changes made to a system to impact the entire program. The complexity of the source impact analysis is polynomial in the number of source lines of code.

4.2 Directed Symbolic Execution

Before we present the details of the second phase of DiSE, we first provide a brief description of symbolic execution.

4.2.1 Symbolic Execution

Symbolic execution [9,19] is a non-standard approach to program execution that uses symbolic values in place of concrete (actual) values for program inputs. The output values are computed as expressions defined over constants and the symbolic input values, and using a specified set of operators. To illustrate symbolic execution, we use the following code fragment:

```
int y;
...
  int testX(int x){
   if (x > 0)
     return y + 1;
   else
     return y - 1;
   }
```

To perform symbolic execution on this code fragment, two symbolic variables are used: Y represents the symbolic value of the integer field y, and X is the symbolic integer value used to represent x, the integer argument to `testX`. During symbolic execution, a path condition is used to collect constraints on the program inputs that will result in execution of the current path. In this example, symbolic execution computes two path conditions: (1) when $X > 0$, the value $Y + 1$ is returned, and (2) when $\neg(X > 0)$, the value of $Y - 1$ is returned. The program behavior summarization would be as follows:

1. $X > 0 \wedge ret == Y + 1$
2. $\neg(X > 0) \wedge ret == Y - 1$

where *ret* indicates the return value of the method.

During symbolic execution, the current path condition is checked for satisfiability. A decision procedure is used to check if there exists an assignment of values to the program variables that will make the constraints in the path condition satisfiable (true). When the constraints on the path condition are not satisfiable, the execution path is marked as infeasible. The execution stops along an infeasible path and the search backtracks. In programs with loops and recursion, infinitely long execution paths may be generated. In order to guarantee termination of the execution in such cases, a user-specified depth bound is provided as input to symbolic execution. Whenever the size of the current execution path reaches this user-specified depth bound, the search backtracks.

At the end of symbolic execution, all of the path conditions generated are collected into a symbolic summary. Each path condition in the symbolic summary represents a set of (feasible) concrete execution paths. The path conditions in a symbolic summary can be used as input to other program analysis techniques, such as regression testing. For example, the values of a solved path condition form the set of concrete input values that will cause the program to execute that path in the program, and as such can be used to generate or select regression tests.

4.2.2 Computing Impacted Path Conditions

The second phase of DiSE performs a form of *incremental* symbolic execution on the modified version of the program. DiSE directs symbolic execution to explore only the parts of the program that are impacted by the changes to the code. DiSE leverages the set of impacted source lines computed in the previous phase, and the reachability information encoded in the CFG as input in order to explore a subset of the feasible execution paths. When no impacted statements are reachable on the current path, symbolic execution backtracks, avoiding the cost of unnecessarily exploring and characterizing execution paths in the modified version of the program that are not impacted by the change(s) to the program.

There is another important aspect of pruning within DiSE that sets it apart from other change-impact analysis techniques. DiSE prunes certain symbolic execution paths by exploring only a subset of the possible choices. DiSE may prune choices at conditional branch statements, e.g., when these statements are not marked as impacted in phase I, even if other impacted source lines are reachable from the block. We demonstrate the intuition for this pruning through an example:

```
int a, b;
  int pruneTest(int x, int y){
1:  if ( x > 0 )
2:     a = x + 1;
3:  else
4:     a = x - 1;
5:  if ( y > 0)*
6:    b = y + 2;*
7:  else*
8:    b = y - 2;*
9:  }
```

Suppose, the source lines of code identified with an $*$ are marked as impacted in phase I. There are two conditional statement blocks, one block at lines $1 - 4$ is controlled by the value of variable x, while the other block at lines $5 - 8$ is controlled by the value of variable y. There are four possible symbolic paths in this program:

1. $(X > 0) \wedge (Y > 0)$
2. $(X > 0) \wedge \neg(Y > 0)$
3. $\neg(X > 0) \wedge (Y > 0)$
4. $\neg(X > 0) \wedge \neg(Y > 0)$

Since the first conditional block is not impacted by the change, DiSE explores only one choice for the value of x, i.e., explores the same path through the unimpacted code ($X > 0$ or $\neg(X > 0)$) for all program executions through the unimpacted code. As a result, DiSE prunes two of the paths shown above, e.g., it prunes 1 and 2, or 3 and 4. Which paths are pruned is determined by the search strategy implemented by the symbolic execution engine, e.g., random, greedy, default.

The resulting set of path conditions computed by DiSE then characterizes the set of program execution behaviors in the modified version of the procedure that are impacted by the change(s). These path conditions serve as the input to the change-impact analysis presented in Section 5.

4.2.3 Scalability and Limitations of DiSE

Scalability The use of symbolic execution to compute impacted program behaviors is the primary factor affecting the scalability of the DiSE algorithm. Recent advances in reduction and abstraction techniques, constraint solving, raw computing power, and in the development of novel reuse techniques such as [43,47], have helped to improve the scalability of symbolic execution. These improvements to symbolic execution can be leveraged to help improve the scalability of DiSE. The smaller symbolic summaries computed by DiSE benefit the program analysis techniques which use the DiSE results by reducing the scope of the analysis to the program behaviors impacted by the differences.

Limitations The DiSE algorithm was originally implemented as an intraprocedural analysis. In [34] we present iDiSE, an interprocedural version of our algorithm. The current versions of the DiSE and iDiSE algorithms do not compute the impact of changes to dynamically allocated data or changes to global data, e.g., fields in Java classes; however, we are working on a version of the iDiSE algorithm to compute the impact of these types of changes.

Other limitations of our change impact analysis are related to the limitations inherent with the use of symbolic execution. In Section 4.2.1 we explain how a user-specified depth bound may be necessary to avoid infinitely long execution paths when the loop bounds are unknown a priori. Other limitations related to symbolic execution include the availability of the underlying theories in the decision procedures used by the symbolic execution engine. For example, to reason about non-linear arithmetic and operations on complex data structures and library operations on those structures. It is interesting to note that these limitations are actually part of the motivation for DiSE – our goal was to avoid the program structures which contribute to these limitations whenever possible by exploring only the parts of the symbolic execution space that is impacted by the changes to the code.

5 Application

In previous work [28,34], we discuss how the results of DiSE can be used to support regression testing techniques and delta debugging techniques. The path conditions generated by DiSE along impacted program statements are solved to facilitate regression testing tasks. We present an evaluation in [28,34] that demonstrates how the solutions to the impacted path conditions can be used for better test case selection and augmenting the existing test suite compared to just using symbolic execution. We also show how the output of DiSE can be configured for generating test inputs that satisfy different coverage criteria, e.g., impacted branch coverage, impacted statement coverage, among others. The information about which constraints are generated at impacted program locations can be used to improve the efficiency of delta debugging as shown in [34]. We have analyzed synchronous reactive components from the automotive as well as the avoinics domain. For example, we have previously analyzed versions of the Altitude Switch (ASW) application that turns power on to a device of interest when the aircraft descents below a threshold altitude above ground level. We have also analyzed NASA's On-board Abort Executive (OAE) that models the Crew Exploration Vehicles' prototype ascent abort handling software.

In this section, we discuss a new application of DiSE, demonstrating its utility in maintaining a software health management framework that uses a Bayesian network. We demonstrate the value of the change-impact information computed by DiSE in facilitating the process of managing the health of the SWHM monitor as the monitored software is changed. We first present background information on Bayesian networks and discuss the advantages of using Bayesian networks for software health management. We then present an example software health management system modeled as a Bayesian network, and describe how the change-impact information about the monitored software can be used to help update and test the software health manager represented as a Bayesian network.

5.1 Bayesian Networks

Bayesian networks are used to reason about data in the presence of uncertainty [11,25]. A Bayesian network is a directed acyclic graph where the nodes in the graph represent statistical variables in the system, and the edges between the nodes represent dependencies between the different variables in the system. Recent work has explored using Bayesian networks to model software health management systems [37–39]. The software health manager monitors various software and hardware systems. Data from the hardware and software sensors is presented as evidence to the nodes in the Bayesian network. Based on the data, the Bayesian network reasons about failures and root causes for the failures in the system (hardware or software) being monitored.

Bayesian networks contain multiple types of nodes as used in this approach for SWHM. Each type of node has a specific role in the system. *Command* nodes receive signals that are interpreted as commands. *Sensor* nodes receive signals that provide data about the variables in the monitored hardware or software. *Health* nodes indicate the health status of a sensor. *Status* nodes encode the unobservable status of a particular sub-system. And, *behavior* nodes connect various nodes in the network in order to recognize behavioral patterns.

Bayesian networks have several advantages for modeling software health management systems. For example, they have full forward and backward reasoning capabilities. In forward reasoning, the network calculates the probabilities on the status of the health nodes based on the values of the sensor nodes; this provides diagnosis and 'most likely' root cause explaining the current data. In backward reasoning, when the network observes a certain diagnosis, it can reason about which sensors are most likely broken.

5.2 Example Program

We use the example source code shown in Fig. 2 to demonstrate the challenges of maintaining the health of a software health manager in the context of evolving systems (in this case, software changes). The code fragment shown in Fig. 2 is a simplified version of the Wheel Brake System (WBS). The WBS is a synchronous reactive component derived from the WBS case example found in ARP 4761 [18,35]. The Java code is based on a Simulink model translated to C using tools developed at Rockwell Collins and manually translated to Java. The goal of this code is to determine how much braking pressure to apply based on the environment. It consists of one Java class and a total of 231 source lines of code. The code shown in Fig. 2 is a simplified version of the Java program.

Two versions of the method `update(int PedalPos, int BSwitch, int PedalCmd)` are shown in Fig. 2. In both versions of the program, the `update` method sets the value of two global variables, *AltPress* and *Meter*, based on the input values of its arguments. The version on the left, Fig. 2(a), is the original version, and the version on the right, Fig. 2(b), is the modified version. The change to the code is on line 9 of the `update` method in Fig. 2(b), where an additional *else* clause is added to the `update` method. This code creates an additional case for checking the value of *PedalPos* and setting the value of *PedalCmd*.

An example Bayesian network for the code example in Fig. 2 is shown in Fig. 3. The sensor nodes for the input variables *PedalPos*, *BSwitch*, and *PedalCmd* are labeled respectively in Fig. 3. The nodes in the Bayesian network labeled with the prefix `H_` are health nodes. For example, *H_PedalPos* is a health node that monitors the health of the sensor node *PedalPos*. The node's probabilities give an indication about the health of the component. The node usually has two states "healthy" and "bad" where the summation of their probabilities is one: p(healthy) + p(bad) = 1. The nodes in the Bayesian network labeled with the prefix *U_* are command nodes that represent update variables in the system. Variables in the system are identified by their node label. The final updates to the global variables flow to the nodes labeled *AltPress* and *Meter* in Fig. 3. The edges between the nodes represent dependencies. For example, the updates to the *PedalCmd* variable in Fig. 2(a) are only possible for certain values of *PedalPos*. The assignment of the value is contingent on the conditional statements at line 5 or 7 in Fig. 2(a) evaluating to true.

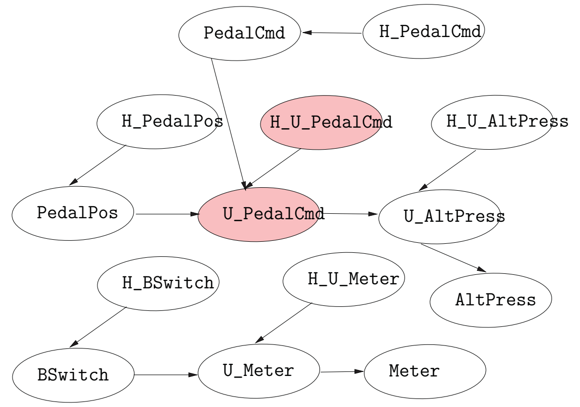

Fig. 3 The Bayesian network for the example in Fig. 2.

For the purposes of maintaining the health of the SWHM monitor, it is useful to identify the areas in the Bayesian network that are not impacted by the changes to the monitored code; these parts of the network do not need to be re-analyzed or re-tested, potentially leading to a considerable savings in the maintenance costs. Even just the basic ability to mark the impacted nodes in the Bayesian network is useful because it facilitates manual inspection of the network. In large Bayesian networks, the impact analysis can be especially useful to detect the parts of the network that are impacted by the changes made to the monitored software.

By visual inspection of the graph in Fig. 3, we can see that there are two disjoint graphs. The change at line 9 in Fig. 2(b) corresponds to the shaded nodes in Fig. 3. Without any additional information, we can infer that all of the nodes in the top graph are impacted by the changes, whereas the nodes in the bottom graph are not impacted by the changes. In the next section, we illustrate how the change-impact results computed by DiSE can be used to mark the subset of the nodes in the top graph of the Bayesian network in Fig. 3 as impacted.

5.3 Running DiSE

The input to DiSE for the example shown is Fig. 2 is the modified program in Fig. 2(b) and the set of modified source lines of code, which for this example is a singleton set: $\{9 : \textbf{else}\ PedalCmd = PedalPos * 1\}$. This additional statement is added to the code in order to cover all of the possible cases within the first conditional block. The set of modified source lines can be efficiently computed using any source-level *diff* tool.

The static analysis algorithm in DiSE is applied to the modified version of the source code. The change to

```
1: /* Global State Variables */
2: int $AltPress := 0$
3: int $Meter := 2$
4:
int update(int $PedalPos$, int $BSwitch$, int $PedalCmd$)
5: if $PedalPos == 0$ then
6:     $PedalCmd = PedalPos + 1$
7: else if $PedalPos >= 1$ then
8:     $PedalCmd = PedalPos + 2$
9:
10: if $PedalCmd == 1$ then
11:     $AltPress = 0$
12: else if $PedalCmd == 2$ then
13:     $AltPress = 1/4$
14:
15: if $BSwitch == 0$ then
16:     $Meter = 1$
17: else if $BSwitch == 1$ then
18:     $Meter = 2$
```

(a)

```
1: /* Global State Variables */
2: int $AltPress := 0$
3: int $Meter := 2$
4:
int update(int $PedalPos$, int $BSwitch$, int $PedalCmd$)
5: if $PedalPos == 0$ then
6:     $PedalCmd = PedalPos + 1$
7: else if $PedalPos >= 1$ then
8:     $PedalCmd = PedalPos + 2$
9: else $PedalCmd = PedalPos * 1$
10:
11: if $PedalCmd == 1$ then
12:     $AltPress = 0$
13: else if $PedalCmd == 2$ then
14:     $AltPress = 1/4$
15:
16: if $BSwitch == 0$ then
17:     $Meter = 1$
18: else if $BSwitch == 1$ then
19:     $Meter = 2$
```

(b)

Fig. 2 Code fragments from a simplified WBS example: (a) original version and (b) modified version.

the assignment of *PedalCmd* does not have any impact on the block of statements at lines $16 - 19$ in Fig. 2(b) because the value of *PedalCmd* is not used (read) at those lines; however, the block of statements at lines $11 - 14$ may potentially be impacted by the assignment to *PedalCmd* at line 9. The value of *PedalCmd* at lines 11 and 13 is used to determine which code is to be executed, i.e., line 12 or line 14. As a result, at the end of phase I of DiSE, the set of impacted statements will include the following statements:

{
9 : **else** $PedalCmd = PedalPos * 1$,
11 : **if** $PedalCmd == 1$ **then**,
12 : $AltPress = 0$,
13 : **else if** $PedalCmd == 2$ **then**,
14 : $AltPress = 1/4$
}

This set of impacted statements will then be used to direct symbolic execution during the next phase of DiSE. Recall that during symbolic execution of the modified version of the code, checks are made to determine if any impacted program statements are reachable from the current program location. This ensures that only the impacted execution behaviors are explored and characterized. Let us consider the part of a path condition generated during symbolic execution along the impacted set of program locations:

$$PedalPos \neq 0 \wedge PedalPos < 1 \wedge (PedalPos * 1) == 1$$

The variable *PedalCmd* is replaced with the value assigned to it at line 9, $PedalPos*1$. The constraint shown above is, however, not satisfiable, i.e., no assignment to *PedalPos* will make the constraint satisfiable. The first two constraints on *PedalPos* essentially specify that *PedalPos* is a negative number which contradicts the final constraint. Similarly another partial path condition generated along the impacted set of program locations is:

$$PedalPos \neq 0 \wedge PedalPos < 1 \wedge (PedalPos * 1) == 2$$

This path condition is also not satisfiable. Based on the results of symbolic execution we can then state conclusively that the change made to the assignment of *PedalCmd* does not impact the assignment to the global variable *AltPress*. Both path conditions generated along the impacted program locations in Fig. 2(b) show that the change made to the assignment of *PedalCmd* does not impact any other part of the program.

5.4 Impact of changes

The results of DiSE can now be used to color the impacted nodes in the Bayesian network. Nodes *PedalCmd* and *U_PedalCmd* are initially marked as impacted by the change in Fig. 3 based on the syntactic diff. The results of DiSE indicate that *no additional nodes* are impacted by the change made to the monitored program. This is a safe estimation of the impact of the change, in the sense that the analysis does not miss marking any impacted nodes. In other words, DiSE is a precise and conservative technique for generating the set of impacted program execution behaviors.

The application of DiSE results to the coloring of the nodes in the Bayesian network is quite useful for manually inspecting the effects of a change to the monitored system. The size of the Bayesian network can be very large when the monitored system is composed of large numbers of variables. In such cases, the coloring of nodes is a helpful tool for visualizing the impact of the changes. When only a small number of nodes is impacted, it is easy to identify the parts of the network that do not need to be re-analyzed and tested in order to check their correctness. This can result in a significant savings while maintaining the health of the monitor.

5.5 WBS Results

In this section, we present a subset of the results of an evaluation of DiSE performed in [28]. Here we present the results for the WBS example to illustrate how changes to the code may impact the program execution behaviors. Note that the entire WBS program consists of one method that is invoked from a *main* method. We refer to the WBS method and WBS program interchangeably in this section. DiSE is implemented as an extension of the Java Pathfinder toolkit [44]. The details of the implementation are described in Section 6. The goal of the evaluation was to answer two research questions: **(RQ1)** How does the cost of applying DiSE compare to full symbolic execution on the changed WBS program? **(RQ2)** How does the number of impacted path conditions generated by DiSE compare with the number of path conditions generated by full symbolic execution?

In Table 1, we list the results of running DiSE and full symbolic execution on each version of the WBS example. For each mutant (changed) version of WBS, we list the number of CFG nodes changed (*Changed*) and the number of CFG nodes impacted by the changes (*Impacted*). We also present the following metrics—the time to perform DiSE and the time to perform traditional symbolic execution of the mutant version as reported by SPF, the number of states explored during execution of each technique, and the number of path conditions generated by each technique in the resulting method summary. The results for DiSE are listed under the subheading *DiSE* and the results for traditional symbolic execution are listed under the subheading *Full Symbc*.

To evaluate DiSE on the WBS program we needed multiple versions of the program. We generated versions of the WBS program by manually creating mutants of the base version (v0) of WBS because multiple versions of the WBS program are not available. When creating mutants, we considered a broad range of changes that can be applied to the code: change location, change type and number of changes. We introduced changes at the beginning, middle and end of the WBS method. We also considered the control structures in the code, and make changes at various depths in nested control structures. Each mutant has one, two or three changed Java statements, resulting in up to nine changed nodes in the CFG for the changed version of the WBS program as shown in Table 1. Versions 1–6 contain a single changed Java source statement, versions 7–11 contain two changed statements, and versions 12–16 contain three changed statements.

5.5.1 Results and Analysis

RQ1 (Cost). In Table 1, we can see that for the majority of versions in the WBS program, DiSE takes considerably less time than full symbolic execution. In many cases, the differences in time is several orders of magnitude. In the versions where the changes to the program do not impact all path conditions (program paths), DiSE takes at most 20% of the time taken by full symbolic execution. In the versions v1, v7, v10, v14, and v15, where DiSE explores the same number of states as full symbolic execution, the time taken by DiSE is 9%–30% longer than symbolic execution. This extra execution time accounts for the overhead of computing the impacted locations and supporting data structures.

RQ2 (Effectiveness). The number of path conditions computed by DiSE varies greatly between the different versions of WBS. In the versions that DiSE generates the same number of states as full symbolic execution, the number of impacted path conditions are the same as the ones generated by full symbolic execution. For most of the WBS versions, there fewer path conditions generated by DiSE than full symbolic execution, e.g., DiSE generates half the number of path conditions than full symbolic execution. There is a reduction in the number of path conditions generated for other versions: v2, v4, v5, v6, etc.

Overall, the comparison demonstrates that DiSE has potential application for detecting and characterizing impacted program behaviors in evolving software. In the WBS program, DiSE correctly identifies and characterizes the subset of path conditions computed by full symbolic execution as *impacted*. In some instances, the change impacted only a small percentage of path conditions, and in others, the change(s) had a much greater impact. When only a subset of the path conditions were impacted by the changes, DiSE is able to consistently compute the impacted path conditions in less time—often several orders of magnitude—than full symbolic execution; when all of the path conditions were impacted by the changes, the overhead incurred by DiSE is between nine and 30% for the WBS mutants.

Version	CFG Nodes		Time (mm:ss)		States Explored		Path Conditions	
	Changed	Impacted	DiSE	Full Symbc	DiSE	Full Symbc	DiSE	Full Symbc
v1	1	39	03:19	02:30	677,976	677,976	24	24
v2	1	7	00:08	02:22	93	677,976	17	24
v3	1	3	00:27	02:41	65,976	677,976	12	24
v4	1	0	00:08	02:44	17	677,976	1	24
v5	7	56	00:23	03:44	59,610	1,317,048	14	24
v6	1	1	00:08	02:44	17	677,976	1	24
v7	1	39	03:07	02:51	677,976	677,976	24	24
v8	8	57	00:29	03:45	59,610	1,317,048	14	24
v9	2	4	00:33	02:41	65,976	677,976	12	24
v10	2	39	03:40	02:51	677,976	677,976	24	24
v11	7	56	00:28	03:43	59,610	1,317,048	14	24
v12	8	65	00:31	03:54	70,129	1,317,048	6	24
v13	9	57	00:29	03:44	59,610	1,317,048	14	24
v14	3	39	03:39	02:51	677,976	677,976	24	24
v15	3	42	03:37	02:51	677,976	677,976	24	24
v16	8	56	00:28	03:43	59,610	1,317,048	14	24

Table 1 DiSE results for WBS

The entire WBS program is 231 lines of Java source code. Since the WBS example is a single method, some changes could impact the entire method. In larger examples, we expect that changes are more likely to be localized to certain methods or components.

The results in this section illustrate the effectiveness of DiSE at characterizing the impact of changes on the execution behaviors of the modified code. In the context of software health management, these results illustrate the potential to reduce the cost and effort of maintaining the correctness of the SWHM monitor when the monitored code is changed by using DiSE.

6 Tool Support

DiSE is implemented within the Java Pathfinder [44] toolkit. It is an extension of the symbolic execution engine, Symbolic PathFinder [24,30].

6.1 Java PathFinder

The Java PathFinder (JPF) model checker is an open-source Java bytecode analysis framework. The core of JPF is an explicit state model checker for Java bytecode. JPF is a customized Virtual Machine that supports state storage, state matching, and configurable execution semantics of bytecode instructions. It supports controlled scheduling choices in concurrent programs, and monitoring of program executions with Observer design patterns. It checks for properties such as deadlock, race conditions, and the absence of unhandled exceptions. One of the defining qualities of JPF is its extensibility. JPF has been extended to support symbolic execution, directed automated random testing, configurable state abstractions, various heuristics for enabling bug detection, configurable search strategies, checking of temporal properties and much more. JPF supports these extensions at the design level through a set of stable, well-defined interfaces.

6.2 Symbolic PathFinder

Symbolic Pathfinder (SPF) is the symbolic execution engine for JPF. SPF is an open-source execution engine that symbolically executes Java bytecode. SPF supports a variety of constraint solvers/decision procedures for solving path conditions such as Choco [8], IASolver [16], and CVC3 [10]. In general, state matching is undecidable when states represent path conditions on unbounded input data. Hence, SPF does not perform any state matching and explores the symbolic execution tree using a stateless search. Furthermore, if the solver is unable to determine the satisfiability of the path condition within a certain time bound, SPF treats the path condition as unsatisfiable. This limitation of the constraint solvers may cause symbolic execution to not generate path conditions for feasible execution paths. Loops and recursion can be bounded by placing a depth limit on the search depth in SPF or by limiting the number of constraints encoded for any given path; SPF indicates when one of these bounds has been reached during symbolic execution.

6.3 Directed Incremental Symbolic Execution

DiSE extends SPF by implementing the custom dataflow and control-flow analyses used to compute the set of impacted program statements. The control- and dataflow analyses compute a conservative approximation of the impacted Java bytecode instructions in changed methods in the modified program. The implementation supports both intra-procedural analysis (data and control flow within a method) and inter-procedural analysis (data and control flow across different method calls). The impacted Java bytecode instructions are used to direct symbolic execution along execution paths leading to impacted Java bytecode instructions, while the other paths are pruned. DiSE is implemented in an extension called `jpf-regression`. The output of DiSE is a set of path conditions that describe the constraints over the input and global variables. These constraints represent the impacted program behaviors of the modified program.

We use the impacted program behaviors to characterize the impacted parts of the SWHM system manually. As part of our future work, we plan to automate this process.

7 Conclusions and Future Work

Software health management techniques monitor deployed software in its execution environment to detect violations, predict possible failures, and to help the system recover from faults. When the monitored software is changed, the SWHM monitor software may also need to change in order to continue to operate correctly. In this work we describe how the results of Directed Incremental Symbolic Execution, a general change impact analysis technique we developed previously, can be used to maintain the correctness of a SWHM monitor when the monitored software is changed. To the best of our knowledge, existing software health management techniques have not addressed the issue of maintaining the correctness of the SWHM monitor over time as the monitored software evolves.

The particular SWHM monitor software analyzed in this work is based on Bayesian Networks. Although we have automated the analysis to compute the impact of the changes on the monitored software, we have not yet automated the process for updating the nodes in the Bayes Network to indicate the impact of the changes. For future work, we plan to automate this step and to apply DiSE to larger programs to empirically evaluate the effectiveness of this approach to maintaining the health of the SWHM monitor software. We also plan to explore how the results of DiSE can be used to support other aspects of software health management, and to apply DiSE results to other SWHM frameworks.

We believe that the core concept of DiSE can be adapted and applied to other SWHM techniques as well. An approach has been described for the formal verification for the diagnostics systems using symbolic model checking [26]. The diagnosis system observes a physical system that is modeled as a Kripke structure. The DiSE algorithm could be adapted to generate the set of affected behaviors on the Kripke structure. The impacted behaviors can then be used to check the correctness of the diagnosis system. There is another model-based prognostic technique that uses a simulation of a system collected under nominal, as well as degraded conditions [22]. DiSE could be adapted to generate impacted simulations of a system based on the changes. The impacted simulations could then be used to help maintain the prognostics system.

Acknowledgements The authors thank Johann Schumann for his valuable insights on Bayesian Networks and how they are used within the software health management domain, and for discussions on how the change impact analysis results can be used within this domain to maintain the health of the monitors. The authors also thank Paul Miner and Ben Di Vito for their helpful comments to improve the paper.

References

1. Al-Khanjari, Z.A., Woodward, M.R., Ramadhan, H.A., Kutti, N.S.: The efficiency of critical slicing in fault localization. Software Quality Control **13**, 129–153 (2005)
2. Apiwattanapong, T., Orso, A., Harrold, M.J.: Jdiff: A differencing technique and tool for object-oriented programs. Automated Software Engineering **14**(1), 3–36 (2007)
3. Arnold, R.S., Bohner, S.A.: Impact analysis - towards a framework for comparison. In: Proceedings of the Conference on Software Maintenance, ICSM '93, pp. 292–301 (1993)
4. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. **1**, 11–33 (2004)
5. Backes, J., Person, S., Rungta, N., Tkachuk, O.: Regression verification using impacted program behavior summaries. In: TACAS (submitted) (2012)
6. Binkley, D.: The application of program slicing to regression testing. In: Information and Software Technology Special Issue on Program Slicing, pp. 583–594 (1999)
7. Buse, R.P., Weimer, W.R.: Automatically documenting program changes. In: Proceedings of the IEEE/ACM international conference on Automated software engineering, ASE '10, pp. 33–42. ACM, New York, NY, USA (2010)
8. Choco: Main–page Choco. —http://www.emn.fr/z-info/choco-solver/— (2010)
9. Clarke, L.A.: A program testing system. In: Proceedings of the 1976 annual conference, ACM '76, pp. 488–491 (1976)

10. CVC3: CVC3 page. —http://www.cs.nyu.edu/acsys/cvc3— (2010)
11. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press, Cambridge, UK (2009)
12. Dubey, A., Karsai, G., Mahadevan, N.: Model-based software health management for real-time systems. In: Aerospace Conference, 2011 IEEE, pp. 1 –18 (2011)
13. Godefroid, P., Lahiri, S.K., Rubio-Gonzalez, C.: Incremental compositional dynamic test generation. Tech. Rep. MSR-TR-2010-11, Microsoft Research (2010)
14. Gupta, R., Jean, M., Harrold, M.J., Soffa, M.L.: An approach to regression testing using slicing. In: ICSM, pp. 299–308 (1992)
15. Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., Gujarathi, A.: Regression test selection for java software. In: OOPSLA, pp. 312–326 (2001)
16. IASolver: IASolver page. —http://www.cs.brandeis.edu/ tim/Applets/IAsolver.html— (2010)
17. Jin, W., Orso, A., Xie, T.: Automated behavioral regression testing. In: ICST, pp. 137–146 (2010)
18. Joshi, A., Heimdahl, M.: Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In: SAFECOMP, *LNCS*, vol. 3688, pp. 122–135 (2005)
19. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)
20. Law, J., Rothermel, G.: Incremental dynamic impact analysis for evolving software systems. In: Proceedings of the 14th International Symposium on Software Reliability Engineering, ISSRE '03, pp. 430– (2003)
21. Law, J., Rothermel, G.: Whole program path-based dynamic impact analysis. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, pp. 308–318. IEEE Computer Society, Washington, DC, USA (2003)
22. Luo, J., Pattipati, K.R., Qiao, L., Chigusa, S.: Model-based prognostic techniques applied to a suspension system. IEEE Transactions on Systems, Man, and Cybernetics, Part A **38**(5), 1156–1168 (2008)
23. Orso, A., Apiwattanapong, T., Harrold, M.J.: Leveraging field data for impact analysis and regression testing. In: Proc. ESEC/FSE-11, pp. 128–137 (2003)
24. Păsăreanu, C., Rungta, N.: Symbolic PathFinder: symbolic execution of Java bytecode. In: ASE, pp. 179–180 (2010)
25. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, San Mateo, CA (1988)
26. Pecheur, C., Cimatti, A., Cimatti, R.: Formal verification of diagnosability via symbolic model checking. In: In Proceedings of the 18th International Joint Conference on Artificial Intelligence IJCAI03, pp. 363–369 (2003)
27. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: FSE, pp. 226–237 (2008)
28. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11, pp. 504–515. ACM, New York, NY, USA (2011)
29. Pike, L., Niller, S., Wegmann, N.: Runtime verification for ultra-critical systems. In: Proceedings of the 2nd Intl. Conference on Runtime Verification, LNCS. Springer (2011)
30. Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: ISSTA, pp. 15–25 (2008)
31. Qi, D., Roychoudhury, A., Liang, Z.: Test generation to expose changes in evolving programs. In: ASE, pp. 397–406 (2010)
32. Raghavan, S., Rohana, R., Leon, D., Podgurski, A., Augustine, V.: Dex: a semantic-graph differencing tool for studying changes in large code bases. In: ICSM, pp. 188–197 (2004)
33. Ren, X., Ryder, B.G., Stoerzer, M., Tip, F.: Chianti: a change impact analysis tool for Java programs. In: ICSE, pp. 664–665 (2005)
34. Rungta, N., Person, S., Branchaud, J.: A change impact analysis to characterize evolving program behaviors. In: ICSM (2012)
35. SAE-ARP4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. SAE International (1996)
36. Santelices, R., Chittimalli, P.K., Apiwattanapong, T., Orso, A., Harrold, M.J.: Test-suite augmentation for evolving software. In: ASE, pp. 218–227 (2008)
37. Schumann, J., Bajwa, A., Berg, P.: Parametric testing of launch vehicle fddr models. In: AIAA Space (2010)
38. Schumann, J., Mbaya, T., Mengshoel, O.: Bayesian software health management for aircraft guidance, navigation, and control. In: Proc. of Conference on Prognostics and Health Management (PHM-2011) (2011)
39. Schumann, J., Mengshoel, O., MBaya, T.: Integrated software and sensor health management for small spacecraft. In: Proc SMC-IT (2011)
40. Strichman, O., Godlin, B.: Regression Verification - A Practical Way to Verify Programs. Springer-Verlag, Berlin, Heidelberg (2008)
41. Taneja, K., Xie, T., Tillmann, N., de Halleux, J.: express: guided path exploration for efficient regression test generation. In: Proc. ISSTA, pp. 1–11 (2011)
42. Taneja, K., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Guided path exploration for regression test generation. In: ICSE, New Ideas and Emerging Results, pp. 311–314 (2009)
43. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: Reducing, rusing and recycling constraints in program analysis. In: ESEC/FSE '12, (to appear) (2012)
44. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2), 203–232 (2003)
45. Xu, Z., Rothermel, G.: Directed test suite augmentation. In: APSEC, pp. 406–413 (2009)
46. Yang, G., Dwyer, M.B., Rothermel, G.: Regression model checking. In: ICSM, pp. 115–124 (2009)
47. Yang, G., Păsăreanu, C.S., Khurshid, S.: Memoized symbolic execution. In: ISSTA, pp. 144–154 (2012)
48. Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L.: How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE '11, pp. 26–36. ACM, New York, NY, USA (2011)